US-PAT-NO: 5475754

DOCUMENT-IDENTIFIER: US 5475754 A

TITLE: Packet video signal inverse transport processor memory address circuitry

DATE-ISSUED: December 12, 1995

INVENTOR-INFORMATION:

NAME COUNTRY	CITY	STATE	ZIP CODE	
Bridgewater; Kevin E.	Indianapolis	IN	N/A	N/A
Deiss; Michael S.	Zionsville	IN	N/A	N/A

US-CL-CURRENT: 380/212, 348/384.1 , 348/409.1 , 348/441 , 370/392 , 370/474 , 380/241

ABSTRACT:

In an inverse transport processor, program component packet payloads of respective program components are directed to select areas of random access memory (RAM) (18) in accordance with a plurality of start and end pointers which are stored in a first plurality (86, 89) of registers, one for each program component. Addresses are generated, in part, by a plurality of read pointer registers (82) multiplexed with an adder (80) to successively increment the pointers for respective program components. The **start pointers are associated with read** pointers to from memory addresses that scroll through designated memory blocks selectively assigned to respective program components. Memory access for read and write functions are arbitrated (98) so that no incoming program data can be lost, and all component processors are serviced.

20 Claims, 8 Drawing figures

Exemplary Claim Number: 20

Number of Drawing Sheets: 5

---------- KWIC ---------

Abstract Text - ABTX (1):

In an inverse transport processor, program component packet payloads of respective program components are directed to select areas of random access memory (RAM) (18) in accordance with a plurality of start and end pointers which are stored in a first plurality (86, 89) of registers, one for each program component. Addresses are generated, in part, by a plurality of read pointer registers (82) multiplexed with an adder (80) to successively increment the pointers for respective program components. The **start pointers are associated with read** pointers to from memory addresses that scroll through designated memory blocks selectively assigned to respective program components. Memory access for read and write functions are arbitrated (98) so that no incoming program data can be lost, and all component processors are serviced.

Brief Summary Text - BSTX (6):

The present invention is a memory arrangement for and inverse transport processor. Program component packet payloads of respective program components are multiplexed to a memory data input port and directed to select areas of random access memory (RAM) in accordance with a plurality of start and end pointers. The start and end pointers are stored in a first plurality of registers, one for each program component. Addresses are generated, in part, by a plurality of read pointer registers multiplexed with an adder to successively increment the pointers for respective program components. The **start pointers are associated with read** pointers to from memory addresses that scroll through designated memory blocks selectively assigned to respective program components. Memory access for read and write functions are arbitrated so that no incoming program data can be lost, and all component processors are serviced.

US-PAT-NO: **5848274**

DOCUMENT-IDENTIFIER: US 5848274 A

TITLE: Incremental byte code compilation system

DATE-ISSUED: December 8, 1998

INVENTOR-INFORMATION:

NAME COUNTRY	CITY	STATE	ZIP CODE	
Hamby; John	Issaquah	WA	N/A	N/A
Gustafsson; Niklas	Bellevue	WA	N/A	N/A
Lau; Patrick	Renton	WA	N/A	N/A

US-CL-CURRENT: 717/153, 717/118 , 719/331

ABSTRACT:

An incremental byte code compiler which provides a high-performance execution environment for dynamically linked languages and for distributed target-independent applications. The execution environment provided by the present invention includes an incremental byte code compiler for generating IL symbols and code objects from a byte code source file, a persistent symbol table for storing the IL symbols and code objects, and an incremental imager for dynamically forming the image of the program from the code objects. The present invention further provides an extremely efficient methodology for dynamically adding program elements to a program under execution.

6 Claims, 23 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 14

---------- KWIC ---------

US Patent No. - PN (1):
5848274

Detailed Description Text - DETX (18):
Code Objects: A code object represents a function or data definition as a sequence of bytes in the form required by the CPU architecture of the target processor. A code object includes a set of relocations, each of which specifies that a **pointer** value within the code object must point to the code object that is the definition of a specific IL symbol in order to load the code object as part of an executable program.

Detailed Description Text - DETX (36):
A principal aim of the incremental builder of the present invention is to minimize the compilation effort and time required to construct a complete compiled representation of a computer program after a change to the program's source text, without requiring a specialized representation of program source (e.g.: program source text implemented as database objects) or excessive

overhead in initial compilation. These shortcomings are evident in the inventions taught for instance by WO 95/00901 and WO 95/00904. The system collectively taught therein requires that program source text be implemented as database objects. Furthermore, while these and other previously developed incremental compilers may provide advantages in recompilation, they are slow when building an entire program using very complicated internal structures and they limit the use of a number of desirable tools that operate on source files. In contrast, the incremental builder of the present invention accomplishes its principal aim by a process referred to hereinafter as minimal recompilation. Because the incremental builder of the present invention does not utilize source text implemented as database objects, it is capable of much faster initial building than previous incremental compilers. Furthermore, on rebuilding a program, its use of **pointers** as opposed to database object Ids results in faster incremental rebuilds.

Detailed Description Text - DETX (95):
Referring now to FIG. 10, the compilation of source file A, 1098, is shown. The building of source file A, 1098, generates a first program unit 1100. Program unit 100 represents the smallest individually compilable element contained within source file A, 1098. The creation of program unit 100 causes the creation of program objects 1110, 1120, and 1130. Program object 1110 represents the declaration of data member X and contains a signature, in this example 101, and a dependents set, whose initial state is empty. The declaration for the function Fred is embodied in program object 1120 also has a signature, in this example 37, and a dependents set which is initially empty. Because program object 1120 represents a function declaration, it will result in the creation of an IL symbol. Accordingly, the **pointer** in the IL symbol field in program object 1120 is set to point to IL symbol 1140. IL symbol 1140 contains **pointers** to an old definition and a new definition both of which are, at this time, set to null.

Detailed Description Text - DETX (99):
Having continued reference to FIG. 11, in this example, program units 1160 and 1170 each create one program object, 1180 and 1190 respectively, and the **pointers** defined in the program objects field of program units 1160 and 1170 point to program objects 1180 and 1190. Program object 1180 is the program object representing the definition of the function Fred previously declared by program object 1120, and as such points to IL symbol 1140 which, it will be recalled, was created responsive to the declaration of the function Fred by program object 1120. In similar fashion, program object 1190 represents the definition of the function Barney previously declared by program object 1130, and again program objects 1130 and 1190 each contain **pointers** to the IL symbol 1150 for the function Barney. This is important to note in that this is the mechanism whereby the present invention achieves linking during the process of compilation. This feature takes advantage of a function commonly found in most compilation systems, that of creating a unique intermediate language symbol to represent one function or piece of static data. Because, however, the present invention is able to track the symbol table across compilation units, it is possible to have one IL symbol for a given function in all compilation units and, therefore, no separate linker is required.

Detailed Description Text - DETX (100):
In forming program unit 1160, the incremental builder detects dependencies on program objects 1110 and 1120. Accordingly, the depends-on set of program unit 1160 includes **pointers** to program objects 1110 and 1120 (as shown along **pointer** line D), and the dependents sets of 1110 and 1120 are each augmented to include a **pointer** to program unit 1160. In similar fashion, program unit 1170 depends on program objects 1120 and 1130. Accordingly, the depends-on set of program unit 1170 includes **pointers** to program objects 1120 and 1130 (as shown along **pointer** line C), and the dependents sets of 1120 and 1130 are each augmented to include a **pointer** to program unit 1170.

Detailed Description Text - DETX (105):
IL symbol 1140 is updated as follows: its old-definition field is set to the prior value of its definition field (i.e. to point to code object 1200) and its definition field is set to point to code object 1220. Program unit 1170 and program object 1190 are not rebuilt. After program object 1180 is rebuilt, IL symbol 1140 is updated as follows: the definition **pointer** is changed from the relocations field of code object 1200 to the relocations field of new code object 1220 which contains the updated object code. Similarly, the old definition field of IL symbol 1140 is reset from the null value to the relocation value of the previously defined code object 1200. All program objects, program units, IL symbols, and code objects are stored in the previously discussed persistent symbol table implemented in a database further implemented in a memory device. For reasons of system efficiency, an object oriented database is utilized in a preferred embodiment of the present invention.

Detailed Description Text - DETX (106):
With respect to FIGS. 10-12, the Dependents field of the program objects (e.g., 1110) and the depends on field of the program units (e.g., 1160) are implemented, in a preferred embodiment, as a binary tree of **pointers** arranged for efficient sorting. For the sake of clarity in these figures, the actual list of **pointers,** and the several **pointers** referred to therein, have been omitted from the figure elements representing program units. The concept of a field comprising a binary tree of **pointers** is well known to those of ordinary skill in the art.

Detailed Description Text - DETX (121):
Addresses are normally used by software by the means of **pointers** (the term used in such programming languages as C, Pascal & C++), or references (used in such languages as Pascal and C++). Both forms simply retain the full, absolute address of an object. While hardware (e.g.: the computer's CPU) also uses addresses, it generally has more complex needs than does software. In order to simplify the logic of the transistors that implement a given hardware architecture, addresses utilized by hardware are formed by a myriad of different schemes, each of which depends on the hardware itself. By way of example, but not limitation, these hardware addresses may be split in two pieces, located in different instructions, or made relative to the address of the instruction that uses the address.

Detailed Description Text - DETX (125):
III. The CPU, i.e. the hardware, which needs to use addresses from category c when accessing data, but may often not be able to use them in the same form (**pointers**) as the Runner software is.

Detailed Description Text - DETX (128):
In the present invention, as in most linker technology, a relocation record is a piece of data that describes what form an address must take at a given point in the program. The form varies not with the referenced address, but with the location where the reference occurs. For example, data items that contain addresses of functions usually need to keep absolute **pointers** to the function, since the address will likely (but not certainly) be used by software.

Detailed Description Text - DETX (137):
5. At address R.sub.b place a **pointer** to S.sub.c +k, where k is the constant offset of the translated object code within the runtime code object. If this **pointer** must be relative to the current location, use R.sub.e +O.sub.L

+k as the current location.

Detailed Description Text - DETX (139):
FIGS. 13 through 17 illustrate a preferred embodiment of incremental image formation according to the principles of the present invention. Referring now to FIG. 13, the use of the code objects and IL symbols in forming the program image is shown. An IL symbol, previously stored in the persistent symbol table, is shown at 2000. Each IL symbol 2000 contains a definition field 2001 which comprises a code object **pointer** 2004 which points to a corresponding code object 2010 by means of direct memory addressing. Definition field 2001 is defined by IL instruction stream 1999. IL symbol 2000 further comprises a dependents field 2002 and an old definition field 2003. Optionally, IL symbol 2000 further defines an IL name 2005. It is important to note that name 2005 is used only to display information to the programmer, e.g., in debug mode, and is defined during image creation. Each IL symbol 2000 is formed by the incremental builder of the present invention from part of the IL stream 1999.

Detailed Description Text - DETX (140):
Code object **pointer** 2004 points to a code object 2010 corresponding to IL symbol 2000. Each code object 2010 contains a size field 2011, a relocations field 2012 and a code section, 2013. Code sections 2013 contain the previously discussed fully translated machine-language implementations of function definitions and the initial values of variables. Relocations field 2012 defines a relocations **pointer** 2014, which points to a relocations array 2020. Relocations array 2020 comprises a vector of relocation offsets and IL symbol references.

Detailed Description Text - DETX (141):
The IL symbol to which relocations **pointer** 2014 points may be the same IL symbol which invoked the formation of code object 2010, as in the case of a recursive process, or in the more typical case, may point to a different IL symbol altogether.

Detailed Description Text - DETX (143):
Referring now to FIG. 14, the formation of run-time code objects is discussed. Each IL symbol 1000 causes the formation of a run-time code object 1010. As the Incremental Imager 200 of the present invention continues to analyze code objects 1010, additional IL symbols 1000 are added to the image. Each of these code objects contains a **pointer** to a relocations table 1020 as previously discussed, which points in turn to an IL symbol 1000, typically of finer granularity, which in turn points to its code object 1010 and so forth. At the point where no more IL symbols 1000 can be added, all the code objects 1020 which define the program have been formed. When all the code objects in the image have been formed, a state of transitive closure is said to exist.

Detailed Description Text - DETX (152):
The debugger directs the imager 200, at step 11, to update the code objects affected by the editing as shown in FIG. 17. Having reference to that figure, IL symbol 901 caused the creation of code object 902 which in turn invoked the creation of runtime code object 903. For the purposes of illustration, in this example, runtime code object 903 is referred to by a second runtime code object 904. After the developer edits that portion of the source code which caused program failure, e.g. an improperly defined function 901, the **pointer** from IL symbol 901 which represents the function is reset from old code object 902 to the new code object 905 which embodies the corrected function. New code object invokes the creation of a new code object 906. To effect this functionality, the **pointer** 910 of old runtime code object 903 is reset from line 0 of its contained code to line 0 of the contained code of new runtime code object 906.